SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract., **10**, 3–35 (1998)

Research

Business Rule Extraction Techniques for COBOL Programs

HAI HUANG[1]*, WEI-TEK TSAI[1], SOURAV BHATTACHARYA[2], XIAOPING CHEN[1], YAMIN WANG[1] and JIANHUA SUN[1]

[1]*Department of Computer Science, University of Minnesota, Minneapolis MN 55455, U.S.A.*
[2]*Department of Computer Science and Engineering, Arizona State University, Tempe AZ 85287–5406, U.S.A.*

SUMMARY

Business rules are operational rules, often coded into software, that business organizations follow to perform various activities, such as transaction processing, quality control, business planning and database management. Over time, business rules evolve and the software that implemented them are also changed and maintained. As the encompassing software becomes large and aged, the business rules embedded are difficult to extract and understand, evolving with the encompassing software over time. Furthermore, the encompassing software is changed without changing the corresponding text documents, and thus, often, the business organization trusts the code more than any other documents.

This paper proposes several techniques to extract business rules from legacy code. It is possible to use a generic software maintenance tool to extract business rules; however this can be an expensive exercise. We propose a tailored solution approach for the business rule extraction (BRE) problem, which combines variable classifications, program slicing, heuristics for identifying slicing criteria, multiple representations of business rules, bottom-up data flow analysis and hierarchical abstraction, among other maintenance techniques. The proposed solution approach has been implemented as a system and successfully tried with a number of industrial programs. © **1998 John Wiley & Sons, Ltd.**

KEY WORDS: business rules; program slicing; data flow analysis; hierarchical abstraction; domain knowledge; code extraction

1. INTRODUCTION

A significant amount of legacy code is used in business applications. These legacy programs usually contain business rules (e.g., material handling, billing regulations, customer interactions, business decisions) that have been coded into the software over the years. These business rules are being maintained as an integral component of the software being maintained. These legacy programs are usually huge and have been developed and maintained over a long period of time. In the course of time, over numerous update

* Correspondence to: Hai Huang, Department of Computer Science, University of Minnesota, Minneapolis MN 55455, USA. E-mail: hhuang@cs.umn.edu

CCC 1040–550X/98/010003–33$17.50

Received 1 July 1997
Revised 2 September 1997

phases, many of these programs have grown so large (sometimes millions lines of code) and complex that they are difficult to maintain.

Telephone billing rates are an example of business rules. A typical long-distance phone call will be charged according to the time of the call, the date of the call (whether it is a holiday, a weekend or a weekday), the time span of the phone connection, the locations of the calling parties and called parties, the type of call (direct-dial calls, conference calls, operator-assisted calls, toll-free calls, collect calls, flat rate calls, etc.), and the specific calling plans in which the calling and called parties are enrolled. Every time a customer makes a call, the billing program of the telephone company needs to keep track of all the relevant information and record appropriate charges according to the applicable billing rate.

Business rules are subject to changes as the markets and technology change. For example, many long-distance phone companies need to change their business rules. This is to retain existing customers and to attract potential customers to change their telephone service providers. When an update occurs in the business practices or rules, the corresponding segments of the software must be changed.

Business rules are usually first written in some text documents (e.g., company policy manuals) before they are coded into software. Over time, as business rules evolve and new functionalities are added, both the text documents and the software become larger and increasingly difficult to understand and maintain. The software analysts and programmers, who initially helped transform the textual policy documents into the corresponding software representation, gradually tend to focus only on the software and lose confidence in the text documents. This creates a maintenance problem. Our interaction with several large software maintenance sites, where millions of lines of COBOL programs have been maintained over many years, illustrates this situation (Tsai *et al.*, 1993; Joiner *et al.*, 1994; Joiner and Tsai, 1997; Tsai, 1997; Sano, Tsai and Rayadurgam, 1997). Figure 1 summarizes this.

In such systems, when a business policy update occurs, the maintainers must understand the relevant business rules that could (or should) instantiate that particular update. As discussed above, since the analysts and programmers have little trust and understanding of the text documents, this is a difficult task. The maintainers require an assisted or automated approach to extract business rules from the code.

We now describe an industrial example. One company designed and deployed a distributed control system nearly two decades ago. The system is a huge commercial success and many copies have been installed. During the last 20 years, the system has evolved functionally and in size. These changes were carried out directly on the software system, as well as on the documents. However, owing to its large size, no single person or group of persons understands both the software and the documents. As a result, when a business rule update occurs, it is difficult for the maintainers to carry it out. Also, the maintainers gradually lose their confidence in their text documents because they are not sure that they have made all the necessary updates. In other words, they trust the code more than they trust the written business rules, but they are not able to extract business rules from the code with confidence either.

In such cases, one of the most frequent software maintenance questions is:

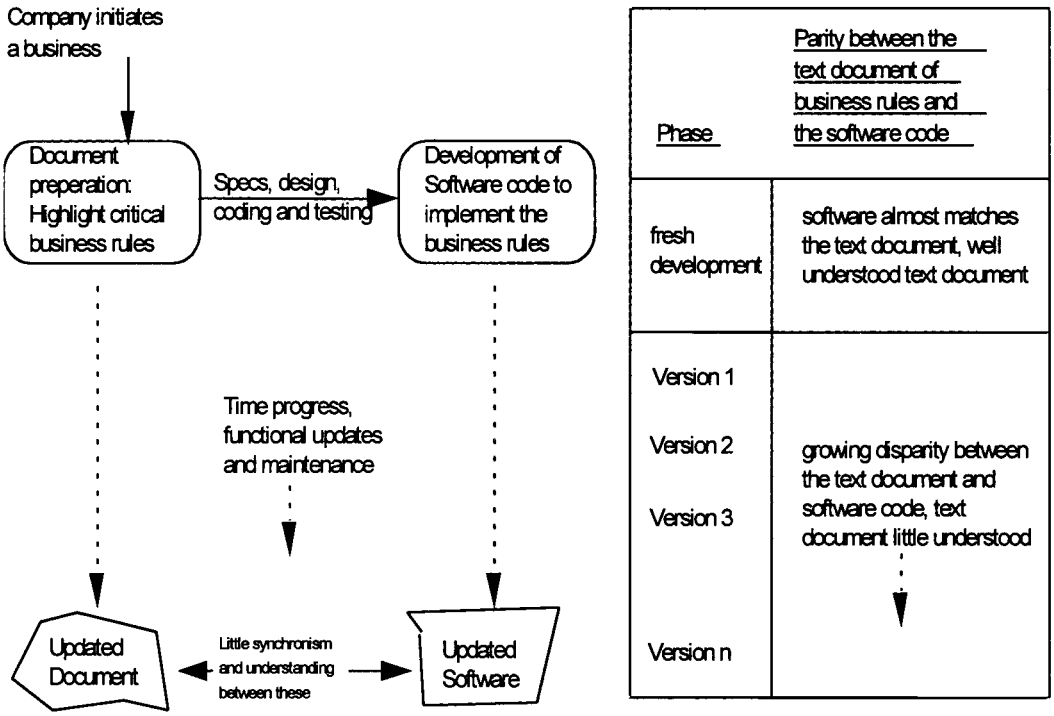


Figure 1. The motivation behind business rule extraction

Is it possible to extract the business rules embedded in the current legacy code so that they can be checked with the rules in the written documents?

This is a critical and important issue because once the business rules are known, it is easier for the business organization to develop new rules by modifying the existing rules. Furthermore, business rules often contain useful proprietary information that might not be available in any other forms due to a variety of reasons, such as the departure of key personnel. Also, the business rules can be used to improve the current software maintenance process and practice.

This paper identifies BRE as an important problem. It is critical for many business operations, and is currently an expensive task for business units that use large amounts of software in their operations. It is possible to use a generic software maintenance tool to extract business rules. We propose a tailored solution approach for the BRE problem, which combines variable classifications, program slicing, heuristics for identifying slicing criteria, multiple representations of business rules, bottom-up data flow analysis and hierarchical abstraction, among other maintenance techniques. The proposed solution approach has been implemented as a system and successfully tried with a number of industrial programs.

2. BUSINESS RULE EXTRACTION CRITERIA

We have interacted about the needs for business rules extraction (BRE) with several software maintenance organizations where large programs have been maintained. From that experience we now summarize their requirements, expressing them as extraction criteria.

Criterion 1: faithful representation. Any business rules extracted from the code must reflect the true state of the software. This is the *most important* criterion, and if this criterion conflicts with other criteria then this criterion should supersede. The reason for the critical nature of this criterion is that programmers trust the code more than they trust the associated documents. If the business rule extracted cannot reflect the true state of the software, there is no reason for the programmers to use the rules extracted. They should just use the code.

Most of the business software we encountered are rather structured. However, occasionally it is possible to identify several large spaghetti modules that have hundreds of GO TO paths within a module of a few hundred lines (Joiner *et al.*, 1994). If the software embedding business rules is unstructured, then the business rules extracted will most likely be unstructured. Most maintainers realize this fact and they do not mind reading unstructured business rules as long as they know what they read is what is implemented in the code, and that the mapping has been faithful from the code to the business rules.

Criterion 2: multiple representation and hierarchical abstraction. Different people require different representations of business rules. Software maintenance managers may wish to have a high-level overview of the software, such as what kinds of business rules were implemented in which software modules. However, a practising software maintainer usually wants a detailed representation of the business rules because the maintainer is responsible in real time for adding/deleting/modifying the code that implements those business rules. Thus, a practising maintainer typically likes to have business rules represented as code segments, while managers may prefer decision tables, decision trees and structured charts.

Business rules should be represented in a hierarchical manner. Business rules are often rather complex because they must meet various constraints, such as from legal, marketing and technology considerations. Business rules also change over time. Most legacy programs we encountered store old business rules as well, the reasons being version upgradability. A customer of an existing program may continue to be served using existing rather than new business rules. Also, often old rules may not be properly or promptly removed from the legacy code even though the rules are no longer needed.

Business rules can be long (even thousands of pages), detailed and multithreaded. Thus it can be extremely difficult to trace business rules without some form of abstraction or decomposition.

Criterion 3: domain-specific business policies. Business rules are different from program annotation. Program annotation includes both application knowledge as well as programming knowledge. However, most software maintainers prefer business rules to be expressed in the domain vocabulary, or more accurately, by using the variables that represent a domain concept in the application. For example, a variable such as I or LOOP-INDEX is often used as a program counter in a loop statement, but the variable INVENTORY-RECORD-IN may represent a domain concept because it represents the current inventory

records used in generating billing reports. Any tool used should provide a means to identify important concepts, including data and algorithms related to business rules from other supporting program entities.

Criterion 4: human-assisted automation. To expedite the software maintenance process, the business rule extraction process should be automated as much as possible. However, our previous work on software reverse engineering (Heisler, Tsai and Powell, 1989; Chen *et al.*, 1990; Chen, Tsai and Chen, 1992; Heisler, Kasho and Tsai, 1993; Ong and Tsai, 1993; Chen *et al.*, 1994; Joiner *et al.*, 1994, Huang, Tsai and Subramanian, 1996; Joiner and Tsai, 1997) has shown that it is extremely difficult, if not impossible, to devise an automatic program understanding tool. This is also true for business rule extraction. A tool should involve human intervention because business rules are complicated and evolve with the encompassing software over the years. Even if automatic extraction were possible, the resulting business rules obtained might not be good enough to be useful for software maintenance. Thus, the software maintainers prefer to have an interactive tool that allows them to extract business rules, simplify their representations and provide linkage to the code, rather than providing a black box tool that pulls out business rules automatically. The system should also suggest hints to assist the software maintainer in extracting business rules.

Criterion 5: maintenance tool. Business rules extracted should be useful in helping other software maintenance activities. Thus, a BRE tool should be a part of a software maintenance tool kit or workbench. The rules extracted should be kept together with the software using the same tool. The tool should provide a mapping from any business rule to its corresponding implementing code segments and vice versa. This capability will allow the software personnel to focus attention on only those segments (and functions) of the software that are relevant to a particular maintenance task.

3. PROPOSED APPROACH TO BUSINESS RULE EXTRACTION

3.1. Business rule defined

In general, a business rule can be defined as a function, constraint or transformation of an application's inputs into its outputs. This is so because a business rule must eventually produce some outputs to its intended users, and it must take some inputs from its users to start the processing. Formally, a business rule R can be expressed as a program segment F that transforms a set of input variables I into a set of output variables,

$$O = F\,(I) \tag{1}$$

For example, *Profit* = (*Revenues* – *Expenses*) is a simple business rule where '*Profit*' is the output, '*Revenues*' and '*Expenses*' are the inputs, and subtraction is the function implemented by program segment F.

In business applications, data play a dominant role. Thus, a business rule is usually centred around certain data, either input or output data. For example, the above simple business rule can be attached either to the output variable '*Profit*', to either or both of the two input variables, or to any combination of these data in the text documents.

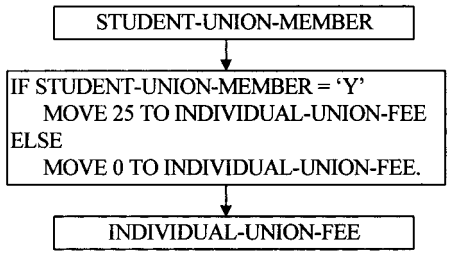


Figure 2. Example of a business rule with its input at the top and its output at the bottom

For another simple example, suppose that INDIVIDUAL-UNION-FEE is an output variable of a program, and that STUDENT-UNION-MEMBER is an input variable used to determine the value of the output variable. The business rule to compute INDIVIDUAL-UNION-FEE is shown in Figure 2, where the first box shows the input, the last the output and the middle the transformation from the input to the output.

A business rule can be used by different people for different purposes, and perhaps with varying degrees of abstraction. For example, a manager may view a business rule as a top-level relationship between input and output variables. Thus, the manager would be mostly interested in the *specification level* of the business rule. On the other hand, a software maintainer may be much more interested in the detailed business rules that are implemented in the code segment because the maintainer needs to have a good understanding of the code before modifying the code. In other words, we need to represent business rules at different levels of abstraction for different users and different maintenance tasks.

3.2. Steps in approach

In this section, we present an approach for extracting business rules from the code, and summarize it in Figure 3. Most of the industrial business software we encountered is

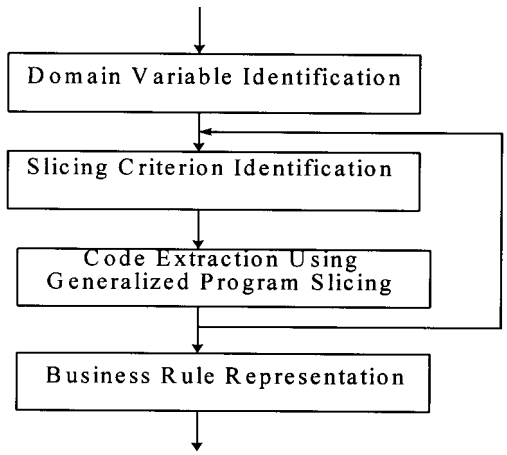


Figure 3. Summary illustration of the business rule extraction (BRE) process

naturally centred on data and their manipulation. Because of Criterion 1, this motivates us to take a data-centred approach (Chen *et al.*, 1994; Joiner *et al.*, 1994; Tsai, 1997) to extract business rules. To fit with expression (1) above, we can first identify either *I*, *O* or both to start BRE. For example, if we wish to extract the business rule that deals with toll-free calls in a telephone billing program, we first identify the variables that represent the toll-free telephone charges, and then extract the code segments that either directly or indirectly manipulate these variables to obtain the business rules related to toll-free charges. This is the data-centred approach to BRE. In contrast, a control–centred BRE approach distils the code segments from the application program first before finding the appropriate data associated with the segment.

The first step in the data-centred approach is to identify important variables from the code that can be used to express business rules. In a typical large business application, we can find thousands or even hundreds of thousands of code variables, but only a subset of them are suitable for expressing business rules—i.e., those that have domain concepts in the application. We call these variables domain variables, and we need a mechanism to identify those domain variables from other code variables. Section 3.3 describes our approach to automatic variable classification.

Once we identify domain variables, we need to select a subset of those domain variables related to a specific set of business rules, and then extract the appropriate code segments that use or are affected by those selected domain variables. The business rules extracted should be expressed in terms of these domain variables only. This is to satisfy Criterion 3 in Section 2. We propose a number of heuristic rules to identify those domain variables in Section 3.4.

Once we have identified domain variables, we extract the relevant code segments by *generalized program slicing* (GPS) (Huang, Tsai and Subramanian, 1996). But, to carry out GPS, we need to specify the slicing criteria which include both the variable names and the starting point in the software. Program slicing is a mechanical technique, and given a set of slicing criteria, an automatic tool can generate the corresponding program slice. However, does the program slice obtained show all the relevant processing in the concerned business rule? Does the slice contain irrelevant or unnecessary data too? Or is the program slice obtained too large to be useful? These issues and approaches to handle them are described in Section 3.5.

Once we obtain the relevant code segments, we need to present the code segments to different users. As indicated in Section 2, different users prefer different representations, and owing to the complexity of business rules, a hierarchical representation of business rules is essential. In Section 3.6, we shall present several presentation schemes to address this issue.

3.3. Variable classification

Variable classification involves categorizing code variables into various types, e.g., domain, program, constants or global variables. This can help program understanding because large programs have hundreds or thousands of code variables, and knowing the role each variable plays in the program, reduces the time to understand the code. Variable classification has been found to be useful for software maintenance (Abbattista, Lanubile

and Visaggio, 1993; Chen *et al.*, 1994). We have proposed an automatic variable classification technique to classify variables from COBOL programs into the following non-exclusive categories (Chen *et al.*, 1994):

- *Domain data*: domain variables are those variables inherent to the application domain and are not dependent on the specific program implementation.
- *Program data*: variables in the program, which are not domain variables, are program data. An example of program data is a variable defined as a COBOL 77 item and used *within* the procedure being considered.
- *Local data*: local variables are data with references confined to a single scope, e.g., one COBOL program.
- *Global data*: global variables are referenced across multiple COBOL programs, or in non-COBOL programs, across multiple procedures. This classification is crucial for programs written in programming languages such as COBOL, where there are few mechanisms to limit the accessibility of a variable.
- *Input data*: input variables are data involved in input events, such as reading from a file or from the keyboard.
- *Output data*: output variables are data involved in output events, such as writing to a file or to the screen. The classification of data as input versus output variables, combined with the classification of domain versus program, gives the maintainer a rich basis for analysing the program.
- *Constant data*: these are variables that do not change in name or value during program execution time.
- *Control data*: control variables are those that are used in predicates (Chapin, 1978). This classification gives the maintainer insight into data purposes in the program. Knowledge of the purpose may be useful if the maintainer wants the program to execute on different paths, as for example during white box testing.

The automatic classification of variables can be done either by parsing the code directly, or by analysing the Module-level COBOL Dependence Graphs (MCDGs) (Joiner *et al.*, 1994) of the code. Most of the classification rules depend on the syntax and semantic of programming languages. Hence, we can devise effective algorithms to classify most variables into the proper categories. Identifying variables as domain variables is different however, because it depends on the application and maintainers, rather than on the programming language used. We have developed heuristic rules to identify domain variables and used the rules to identify domain variables for a sample of industrial code. The variables identified by the heuristic rules are 92% the same as those identified by the maintainers (used as the validation basis). We have implemented all the variable classification rules in a software tool (Chen *et al.*, 1994).

3.4. Selection of domain variables of interest

Once we know how to classify variables into different categories, we need to identify the relevant domain variables among many domain variables in the code. In this section, we describe our two heuristic rules for choosing domain variables of interest.

Heuristic rule 1

We select the overall system's input and output variables as domain variables.

Inputs and outputs are the major characteristics of a software system. The system can be viewed as a black box that maps its inputs to outputs. Thus, identifying input and output variables is a good starting place to explore the system. By tracing input variables to the output variables of the overall system, and vice versa, a maintainer can understand the input to output transformations.

Heuristic rule 2

We select the inputs and outputs of each procedure as domain variables.

Procedures are the functional components of a system. In COBOL software, unlike some other languages, procedures may be implemented in COBOL as statements, sentences, paragraphs, sections and programs, or portions of them. Most of the business applications we encountered are naturally hierarchical in organization, even though not necessarily well-structured. Where hierarchical structure exists, we try to take the advantage of it in BRE. Often, the processing to produce the system outputs from the system inputs is rather involved, and tracing and representing it can be tedious and challenging. Intermediate variables, such as procedure inputs and outputs can simplify it and make it more readable.

Example 1

We now illustrate the selection of domain variables using an example from Grauer (1985, pp. 107–109). This is a program that computes college tuition fees. According to Heuristic Rule 1, we should use the input and output variables of the entire system as the starting point. The inputs to the system are STU_CREDITS, STU_SCHOLARSHIP, IND_UNION and IND_ACTIVITY_FEE, and the output is TOTAL_IND_BILL. The identification of these variables can be done manually or by using the tool described in Chen *et al.* (1994). We can start extracting business rules starting at the input and output statements of the program. In this particular example, we did this by tracing from the output statement backwards, and we obtained the following relationship, where the $*$ indicates multiplication and has precedence over both addition and subtraction:

$$\begin{aligned}\text{TOTAL\_IND\_BILL} = \Sigma\, (&\text{STU\_CREDITS} * 127{\cdot}50 + \text{IND\_UNION\_FEE} \\ &+ \text{IND\_ACTIVITY\_FEE} - \text{STU\_SCHOLARSHIP})\end{aligned} \tag{2}$$

The above business rule simply says that the total tuition bill is the sum of the individual student bills. An individual student bill is the sum of the individual tuition, union and activity fees minus any scholarship available. An individual tuition fee is determined by multiplying a student's number of student credit hours by \$127·50.

Heuristic Rule 2 suggests selecting the input and output variables of each procedure as domain variables. If in this example COBOL program, we treat COBOL sentences as

procedures, then the intermediate variables IND_BILL and IND_TUITION are regarded as domain variables. Then, the business rule can be represented hierarchically as follows:

$$\text{TOTAL\_IND\_BILL} = \Sigma\ (\ \text{IND\_BILL}\ ); \tag{3}$$

$$\text{IND\_BILL} = \text{IND\_TUITION} + \text{IND\_UNION\_FEE} + \text{IND\_ACTIVITY\_FEE} - \text{STU\_SCHOLARSHIP}; \text{ and} \tag{4}$$

$$\text{IND\_TUITION} = \text{STU\_CREDITS} * 127{\cdot}50 \tag{5}$$

3.5. Code extraction using generalized program slicing

3.5.1. Program slicing

Once we know how to select domain variables for extracting business rules and how to select a starting point for tracing code segments, we can either trace the code manually or use a software tool to retrieve relevant code segments. The objective is to retrieve only the code segments that are directly and indirectly affected by and affect the concerned variables. Program slicing can be used for this.

Program slicing and its extensions (Weiser, 1982, 1984; Horwitz, Reps and Binkley, 1990; Gopal, 1991; Chen *et al.*, 1996; Huang, Tsai and Subramanian, 1996) are techniques for automatically decomposing programs by analysing their data flow and control flow. Starting at a given point of the program, program slicing automatically retrieves all the relevant code using data and/or control dependence. Note that program slicing techniques alone are *not* able to obtain business rules because slicing is just like a search engine that will not run until an input, the slicing criterion, is fed to it. Thus, the code extraction should be divided into two steps: (1) identification of the slicing criterion, and (2) performance of the program slicing. Also, we often need to reduce the search space for program slicing.

3.5.2. Identification of slicing criterion

Traditionally, a slicing criterion of a program P is a pair $<i, V>$ where i is a program statement in P and V is a set of variables of concern at statement i. But the pair $<i, V>$ alone is not sufficient for business rule extraction because it places no constraints on the search space. Hence program slicing often produces slices too large to be useful (Huang, Tsai and Subramanian, 1996; Wang *et al.*, 1996). Furthermore, a business rule may span only a particular portion of a program, for example, in one procedure from the acceptance of some inputs to the generation of some outputs. But without constraints, program slicing will search the whole program and produce slices that contain irrelevant or unnecessary code. Hence, a slicing criterion must be generalized to accommodate constraints so as to restrict the search space for slicing.

Thus, identification of the slicing criterion consists of three parts: (1) variable set V; (2) program statement i; and (3) constraint C. Note that V is decided upon by the rules in Section 3.4. We now offer rules for the other two parts. Program statement i may be chosen using the following three heuristic rules.

Heuristic rule 3

The input and output statements of the program are good candidates for starting BRE.

The input and output statements of a program usually indicate the beginnings and the ends of certain processing. By tracing from the input (or output) statements forward (or backward), we will be able to retrieve all of the relevant code segment.

Example 2

Consider a COBOL WRITE statement as below:

```
WRITE MOU-PRINTLINE FROM MAIN-LINE1 AFTER ADVANCING 1 LINES
```

We can start backward slicing at the above statement. The slice we get is a business rule that governs the generation of MAIN-LINE1.

When the processing directed by the procedure between the input and output variables is complex, we may use some intermediate variables to simplify the business rules. It is important to note that intermediate variables *should* consist of domain variables only. Otherwise, the expression of the business rule can become complicated. For this, we use the following heuristic rules to select intermediate variables.

Heuristic rule 4

A location which is a *dispatch centre* in the program is a candidate starting point for forward slicing.

A dispatch centre delegates input data of different types to the corresponding processing procedures. Given an input, the processing originating from the dispatch centre carries the business rules that regulate the processing of inputs of different types. Figure 4 illustrates this situation.

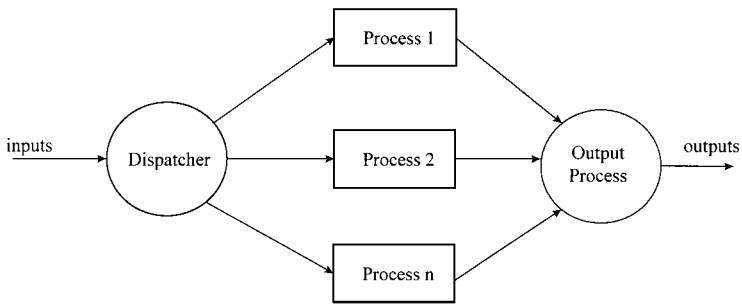


Figure 4. Diagram of a dispatch centre in a program

Example 3

Consider the following COBOL procedure excerpted from a program we observed in use in industry. It dispatches input records to the COBOL sections E01-PROCEDURE, E11-PROCEDURE, E02-PROCEDURE or E04-PROCEDURE in terms of the value of the CARD-NO variable (field) in the record. Clearly, if we start forward slicing from line 055000, 055200, 055400 or 055800, we are able to separate the program into four parts, where each part handles just one kind of input record.

```
054600 PROCESS-A-RECORD       SECTION.
054700 PROCESS-A-RECORD-010.
054800     EVALUATE CARD–NO
054900     WHEN 'E01'
055000          PERFORM E01-PROCEDURE
055100     WHEN 'E11'
055200          PERFORM E11-PROCEDURE
055300     WHEN 'E02'
055400          PERFORM E02-PROCEDURE
055500     WHEN 'E03'
055600          CONTINUE
055700     WHEN 'E04'
055800          PERFORM E04-PROCEDURE
055900     WHEN 'E05'
056000          CONTINUE
056100     END-EVALUATE.
056200     PERFORM READ-INPUT.
056300 PROCESS-A-RECORD-EXT.
056400     EXIT.
```

Heuristic rule 5

An end point of a procedure is a candidate starting point for backward slicing.

Some procedures are dedicated to producing the outputs. There can be multiple occurrences of output events for the same output variable. The sequence of these occurrences of outputs is a business rule that shows how to organize the outputs to form a file, a report, a table or a screen display. Starting from the end of the procedure, a backward trace along all possible paths can identify all the occurrences of the output events with respect to a given output variable and thereby, form a slice. Such a slice is a business rule.

Next, we discuss the constraints on the slicing search space. Two types of constraints on slicing are useful for BRE, depth and boundary.

Depth limit

The depth limit is a constraint which limits the search space by dependence distance. A depth limit can effectively reduce the search space on a dependence graph, and hence, reduce the size of a slice.

Example 4

We conducted a trial on an industrial COBOL application from a telecommunications company using our generalized program slicing tool. This application consists of 11 programs amounting to 15K lines of code. We selected some random points in some programs, and then collected data for the size of slices without setting a depth limit and the size of slices with one as the depth limit. The average size of complete forward slices was 191·8 statements while the size with depth set to one was 3·6 statements. The average size of complete backward slices was 103·3 statements while the size with depth set to one was 4·7 statements.

Boundary

A *slicing boundary* is a set of points in a program that specify a program region to be sliced. These program points are implemented as a set of marks on the nodes in a control flow graph (CFG), such as a call tree.

Program understanding is incremental. A programmer usually focuses on a particular portion of code at one time. The programmer may already understand some functions within this portion, and these functions can be treated as intrinsic components that need not be analysed any more, and can thus be put outside the boundary of the slice.

Example 5

Figure 5 shows a control flow graph of a program. The execution of the program flows from top to bottom and from left to right. The slicing boundary specifies three nodes on the control flow graph. Node A is a start node specifying current slicing focus, nodes B and C are end nodes indicating the known parts within the focus. Given these boundaries,

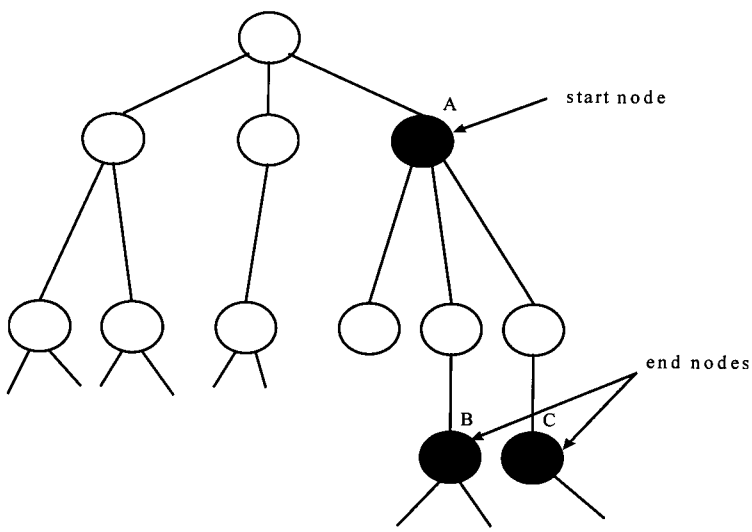


Figure 5. An example of a slicing boundary shown on a call tree

anything outside the subtree from A to B and C is not included in the slice, nor is anything that is under the subtrees headed by B and C.

3.5.3. Choosing slicing algorithms

Selecting a slicing algorithm depends on the chosen slicing criterion and the structure of the code. For this, we offer three heuristic rules.

Heuristic rule 6

Choose forward slicing if the variable in the slicing criterion is an input variable.

Heuristic rule 7

Choose backward slicing if the variable in the slicing criterion is an output variable.

An input is a starting point of dependence, and hence choosing backward slicing does not make sense (see Figure 6). Similarly, since an output is an end point of dependence, forward slicing from an output variable is nearly always useless.

An input variable may be used to produce several different outputs as illustrated in Figure 6. Suppose we select Output_1 and do backward slicing. The produced program slice is only the part of code that affects Output_1. However, if we start from the input and use forward slicing, we will get the whole program.

Heuristic rule 8

Choose forward slicing if the starting program location is a *dispatch centre*.

Since a dispatch centre relegates inputs with different processing units based on the type of inputs, forward slicing can efficiently separate out the processing associated with any particular type of input.

3.5.4. Multiple-pass analysis and recursive slicing

It is not sufficient to have only backward and forward slicing algorithms because business rules are rather complex. Interactive and iterative slicing, in addition to single-pass slicing, may be needed to untangle convoluted processing.

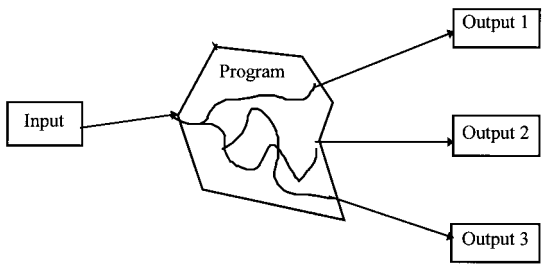
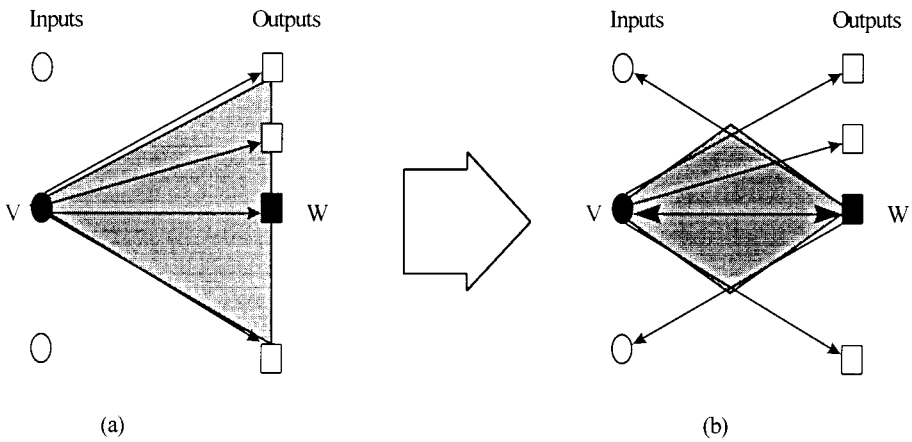


Figure 6. Diagram of a program with one input and three outputs

Recursive slicing (Huang, Tsai and Subramanian, 1996) allows previously obtained slices to be further analysed and decomposed. Traditional program slicing can slice programs only, but slices consistent with the heuristics we propose, need not be programs. Recursive slicing facilitates interactive and iterative program understanding. For example, recursive slicing can be used to sort out the complex relations between multiple inputs and multiple outputs. Usually, one input contributes to more than one output and an output depends on more than one input. Recursive slicing can be used in such a way that we first perform forward slicing, starting from an input to find all the code affected by the input, and then, choose an output to perform backward slicing on the slice. The resulting slice is the code that relates a specific input to a specific output. Figure 7 illustrates this.

3.5.5. High-level abstraction and hierarchical slicing

Traditional program slicing extracts business rules at program statement level. This is inappropriate for large systems. According to Criterion 2 discussed in Section 2, it is desirable to extract business rules at high levels of abstraction, such as via call graphs or module dependence graphs represented at the program level. Module-level and procedure-level slicing can be used to extract relations between modules and procedures. For example, a module slice consists of modules and inter-module variables through which the modules interact with each other. However, it is up to the user to determine which level is most appropriate to use to derive the slice under different situations.



(a) The shaded area is the forward slice from input V;
(b) The shaded area is the recursive slice produced by backward slicing on the previous slice, using the output w as the starting point.

Figure 7. An illustration of recursive slicing

3.6. Presentations of business rules

3.6.1. Three views

At this phase the program slice is ready, with the domain input and output variables as the extreme end–points of the slice. Now, the objective is how to view this slice structure—i.e., what format of presentation for the business rules is most meaningful for the user. We identify three broad choices of view:

- *Code view*. In this format we represent a business rule as code fragments. The code-level view is the lowest representation for business rules. Since software maintainers trust the working code more than any document, and since during software maintenance they prefer a detailed view of business rules, the availability of the code view is essential. The code view can be obtained by pretty printing the program slice obtained earlier.
- *Formula view*. Some users find it desirable to view business rules implemented in the code as *formulae* instead of as code segments. Formulae can be used to annotate the functionality of program slice. A formula can be derived by traversing a dependence graph representing the program slice. During the graph traversal, algebraic substitution is performed until domain variables or control branches are reached.
- *Input–output dependence view*. A program slice is a procedure that links inputs to outputs. An input–output dependence view provides a way to trace data flows either from inputs to outputs or from outputs to inputs. It characterizes the data–flow relationships among all the variables involved in a program slice. It presents business rules in an abstract form. An input–output dependence view of a program slice can also be derived by tracing the corresponding dependence graph.

Note that a program slice corresponds to a subgraph of the program dependence graph (Hecht, 1977; Kuck *et al.*, 1981; Ferrante, Ottenstein and Warren, 1987; Horwitz, Reps and Binkley, 1990; Joiner *et al.*, 1994). To integrate the multiple views of program slices, program slices can be internally represented as dependence subgraphs. Such an internal representation of program slices provides a way of facilitating cross-reference among different representations.

3.6.2. Formula representation of business rules

Next we describe how to derive a formula from a program slice. A business rule can be represented as a set of formulae. A formula has three parts:

- Left-hand side—a domain variable;
- Right-hand side—an expression for determining the left-hand side; and
- Condition—the condition or circumstances under which the formula holds.

Example 6

Table 1 shows an example of a formula representation of a program slice. The left column lists the program slice for the domain variable DET-IND-BILL corresponding to the detailed individual tuition bill. The right column lists the formulae derived from the program slice. Since the variable IND-ACTIVITY-FEE has two possible values depending on the value of the variable STU-CREDITS, two additional formulas are introduced to compute IND-ACTIVITY-FEE as an intermediate variable.

3.6.3. Derivation of formulae from program slices

Formulae can be derived automatically. Suppose $DEF(i)$ denotes the variable defined at statement i, $REF(i)$ the set of variables referenced at statement i and $RD(x, i)$ the reaching definitions (Aho, Sethi and Ullman, 1986, pp. 585–722) of some variable x in $REF(i)$. Let $DEF(i) = f(REF(i))$ denote the semantics of statement i. If $|RD(x, i)| = 1$ and $x = g(.)$ we substitute all the x's in $f(REF(i))$ with $g(.)$. Otherwise, we treat x as an intermediate variable and introduce an additional formula for each reaching definition in $RD(x, i)$. We repeat the above process until no more substitutions or additional formulae can be introduced.

Example 7

Refer again to the program slice in Example 6. Table 2 illustrates how to apply the above process to obtain the formulae listed in the Table 1.

The formula derivation process described above supports the hierarchical representation of business rules. There can be too many combinations if we substitute for all the intermediate variables with domain input variables or constants. Moreover, some intermediate variables are meaningful domain variables too. It is easier to understand formulae

Table 1. Example of formula representation of a program slice

Program slice	Formulae
105 MOVE 25 TO IND-ACTIVITY-FEE. 106 IF STU-CREDITS > 6 107 MOVE 50 TO IND-ACTIVITY-FEE. 108 IF STU-CREDITS > 12 109 MOVE 75 TO IND-ACTIVITY-FEE. 122 ADD IND-TUITION IND-UNION-FEE 123 IND-ACTIVITY-FEE 124 GIVING IND-BILL. 125 SUBTRACT STU-SCHOLARSHIP FROM 126 IND-BILL. 132 MOVE IND-BILL TO DET-IND-BILL.	DET-IND-BILL = IND-TUITION + IND-UNION-FEE + IND -ACTIVITY-FEE − STU-SCHOLARSHIP IND-ACTIVITY-FEE = 50 if STU-CREDITS > 6 IND-ACTIVITY-FEE = 75 if STU-CREDITS > 12 IND-ACTIVITY-FEE = 25 otherwise

Table 2. An illustration of formula derivation

Ref(I)	*RD(x,i)*	*Action*
REF(132) = {IND-BILL}	RD(IND-BILL, 132) = {122}	Substitution: DET-IND-BILL = IND-BILL = IND-TUITION + IND-UNION-FEE + IND-ACTIVITY-FEE − STU-SCHOLARSHIP
REF(122) = { IND-TUITION, IND-UNION-FEE, IND-ACTIVITY-FEE STU-SCHOLARSHIP}	RD(IND-ACTIVITY-FEE, 122) = {105, 107, 109}	Add new formulae: IND-ACTIVITY-FEE = 25 IND-ACTIVITY-FEE = 50 if STU-CREDITS > 6 IND-ACTIVITY-FEE = 75 if STU-CREDITS > 12

arranged in a hierarchy with domain intermediate variables than a single big and complicated formula.

Since program slices can be internally represented as subgraphs of a dependence graph, the formula derivation process can be implemented using a graph searching algorithm. Let *G* denote the dependence graph representing the program slice *S*. Let *Q* denote a queue structure that keeps track of some nodes in *G*. Figure 8 lists a generic algorithm for formula derivation from a dependence graph.

4. IMPLEMENTATION

4.1. DPUTE components

We developed a software tool, called DPUTE for Data−centred Program Understanding Toolset (Joiner and Tsai, 1993; Joiner *et al.*, 1994; Chen *et al.*, 1994; Huang, Tsai and Subramanian, 1996; Wang *et al.*, 1996) to support COBOL program understanding and business rule extraction. In this section, we describe the design and implementation of a prototype of DPUTE that implements the integrated data-centred approach described in Section 3. DPUTE is necessary for verifying techniques and algorithms, and for trials to validate heuristics for variable classification. A block diagram of DPUTE appears in Figure 9. COBOL source code is parsed to create a data structure model and a program dependence model that are stored and later read by intermediate analysis tools in response to queries made by end-users through a graphical user interface. We have implemented this prototype of DPUTE and tried it on two host platforms: UNIX and IBM OS/2. The Unix version is implemented using GNU Flex, Bison and C/C++. The IBM OS/2 version is implemented using GNU Flex, Bison and IBM C Set ++. At present the toolset with its graphical user interface (GUI) consists of seven tools: a parser, a program slicing tool, the REA tool, a variable classification tool, a dependence browsing tool, a formula


```
Add all sink nodes in G to Q
while there is some node in Q do
    Remove the first node x in Q
    Create a formula f: DEF(x) = f(REF(x))
    while there is some variable in REF(x) do
        Remove one variable v from REF(x)
        if there are more than one incoming data dependence arrows to v then
            Add all the source nodes of the arrows to the Q
        else–if there is exactly one incoming data dependence arrow to v then
            Let y denote the source node of the arrow
            Let v = DEF(y) = g(REF(y))
            Substitute all v's in f(REF(x)) with g(REF(y))
            REF(x) = REF(x) U REF(y)
        end–if
    end–while
end–while
```

Figure 8. A generic algorithm for formula derivation from a slice

derivation tool and a control flow analysis tool. Since the REA tool (Yau, Collofello and MacGregor, 1978) tool is not used in BRE, we do not cover it further in this paper.

4.2. COBOL Parser

The parser reads a COBOL source file, builds the data structure model and the program dependence model. The parser includes a preprocessor that removes comment lines and joins continuation lines before beginning the actual parse. The parser is implemented using compiler generation tools based on Lex and YACC. Figure 10 illustrates the phases of the parser.

The front-end analyser includes lexical, syntactical and semantic analysers. It takes as input a COBOL source program and produces a symbol table and a syntax tree. The data-flow analyser performs data-flow analysis on the syntax tree and symbol table, and produces a dependence graph of the program. In this phase, only statement-level and module-level COBOL dependence graphs (SCDG and MCDG) are constructed. These dependence graphs are then stored in databases.

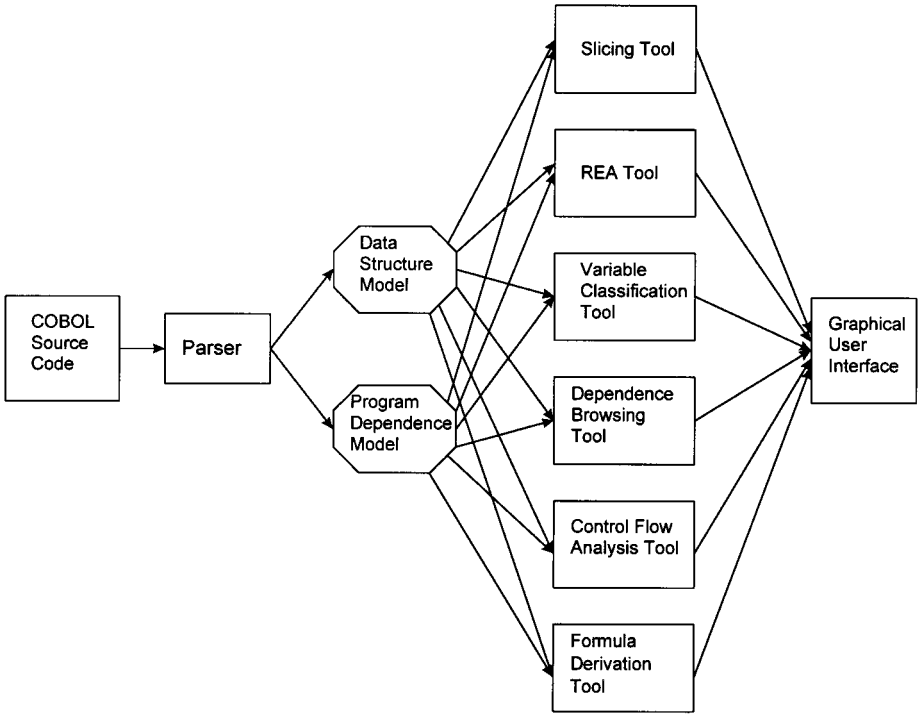


Figure 9. Block diagram of the software tool DPUTE

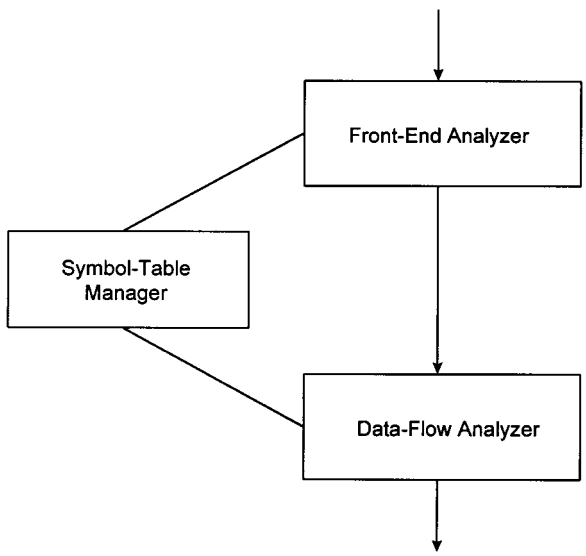


Figure 10. Diagram of the phases of the DPUTE parser

4.3. Variable classification tool

The variable classification tool (Chen *et al.*, 1994) allows users to display classifications for all variables. Classification categories include domain/program, input/output, local/global, constant and control. Users are allowed to override the domain/program classification only; then sensitivity analysis reclassifies the remaining variables.

Figure 11 shows the six components of the module-level tool. They are MCDG information, classification, reclassification, classification table, query and user interface. The MCDG information component takes the command from the user interface, and extracts and feeds the information to the classification component. That component classifies the variables based on the variable classification rules, and writes the classification results with the variable relations in the table. The reclassification component is used when a variable is needed to be reclassified, such as after a user override. The classification table component stores and manages the classifications of all variables and their relations. The query component supports the retrieval of all kinds of data stored in the database. It accepts commands, such as 'list domain variables', from the user interface to search for relevant data, and then displays those data on the screen or writes the data in a file. Figure 12 shows an example of a variable classification user interface screen.

4.4. Program slicing tool

The program slicing tool allows users to perform slicing in either a forward or backward direction starting from a given vertex in the dependence graph. The features of the slicing tool include hierarchical slicing (file level, module level and statement level), constrained slicing allowing users to specify a program region to be sliced, and recursive slicing allowing users to perform slicing on a previous program slice. The tool also supports the other three different varieties of slicing: variable slice, domain slice and condition slice. The output of a variable slice consists of modules and variables that are in the slice. The output of a domain slice consists of only the domain variables of a variable slice. The output of a condition slice consists of the conditions along a slice path.

The program slicing tool is comprised of three components (see Figure 13):

- starting point locator—this maps the slicing criterion to vertices in the dependence graph;

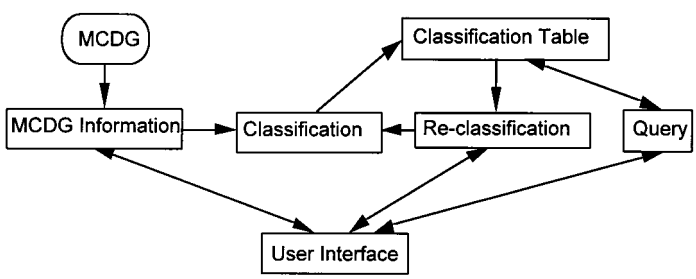


Figure 11. Diagram showing the components of the variable classification tool

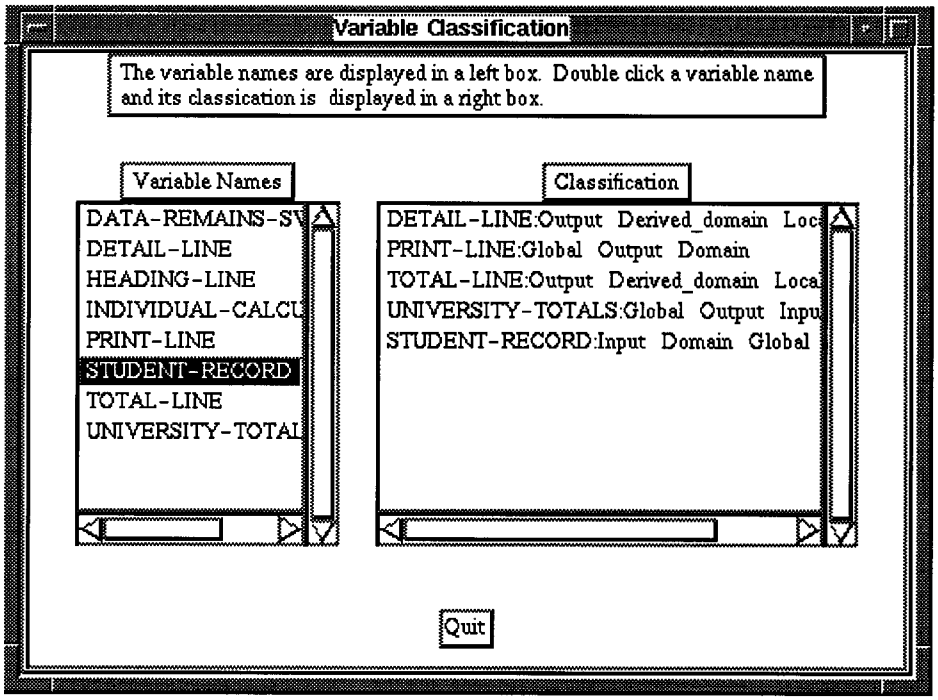


Figure 12. An example of a variable classification user interface screen

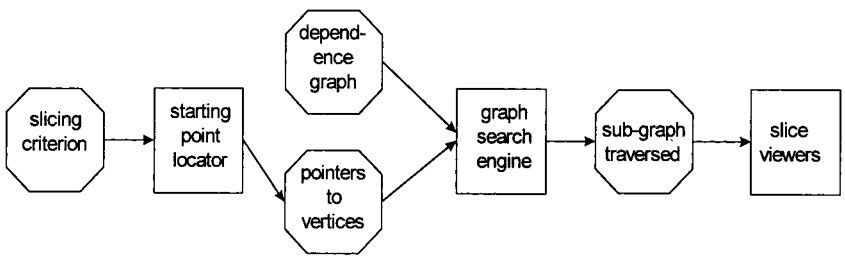


Figure 13. Block diagram of the slicing tool

- graph search engine—this traverses the dependence graph from a given set of vertices and replicates the subgraph traversed where the subgraph represents the slice (the subgraph can also be stored in a database, if requested); and
- slice viewer—this presents, in various views, the subgraph representing the slice, such as text views, dependence graph views and table views.

The text view is the most common representation of a slice. The textual slice viewer takes the dependence graph representing the slice as input and uses the line number data stored at the vertices of the graph to extract the text from the source code. The graphical slice viewers are presented in the next section.

4.5. Graphical dependence browser and slice viewer

The two graphical dependence browsers in DPUTE display a dependence graph as a tree with the starting vertex as the root of the tree. Both browsers are also graphical slice viewers since a slice is internally represented as a subgraph of a dependence graph.

One browser is a statement dependence browser which shows the dependence between statements. In Figure 14, each node is labelled with the line number corresponding to the line in the source program. Each edge is labelled with either a character 'c' or 'd', representing 'control' or 'data' dependency. If the 'Expand Node' button is clicked to 'on', a node can be expanded by clicking over it. If the 'Collapse Node' button is clicked to 'on', the subtree rooted at a node can be removed by clicking it. Similarly, if the

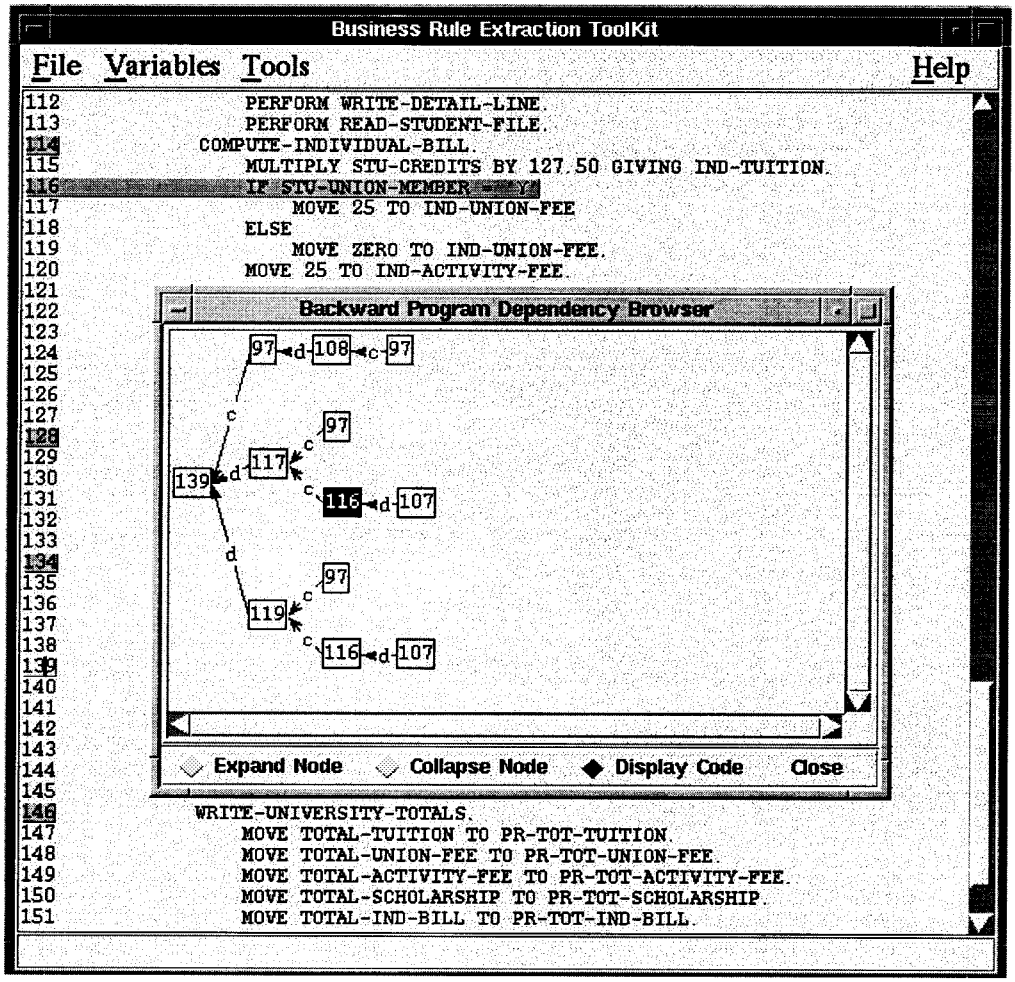


Figure 14. Example of a screen from the statement dependence graphical browser

'Display Code' button is clicked to 'on', clicking a node highlights the corresponding source code in the source code window.

The other browser is a variable dependence browser that displays the dependence between variables. In Figure 15, each node is labelled with a variable name and the line number where this variable occurs. Each edge is labelled with either 'def', 'use' or 'ctrl', representing the 'define', 'use' or 'control' relationship respectively.

Figures 16 and 17 show the graphical browser for MCDGs. Figure 16 is the top–level MCDG for the tuition program without expanding module nodes. The left window in Figure 17 shows the MCDG for the paragraph COMPUT-INDIVIDUAL-BILL, and the right window displays its textual description.

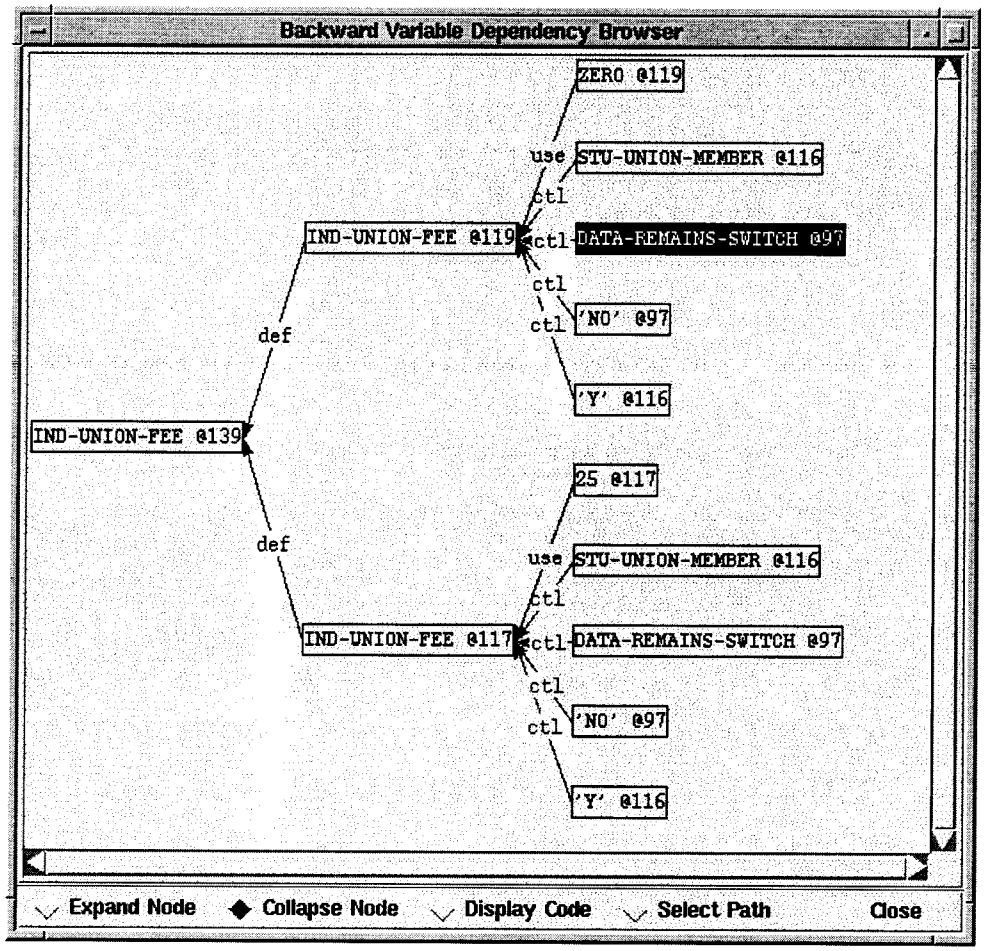


Figure 15. Example of a screen from the variable dependence graphical browser

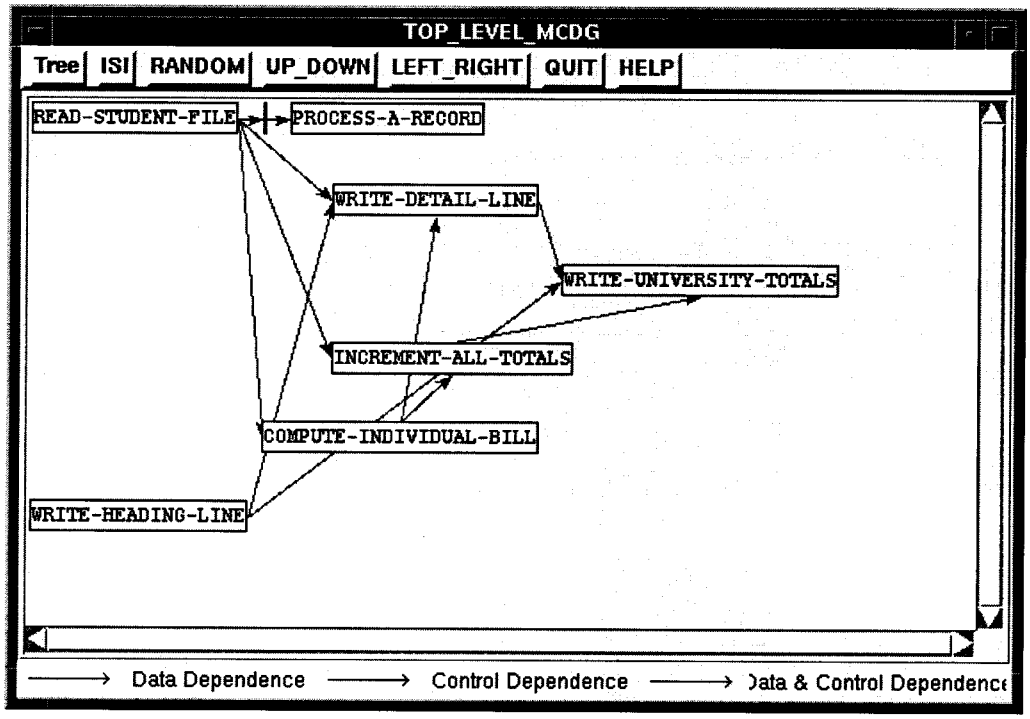


Figure 16. Example of a screen from the MCDG browser for a program

4.6. Formula derivation tool

The formula derivation tool allows user to display formulae or conditions that are used in the processing of any variable involved in the program slice. Users may start the formula derivation tool either from within the dependency browsers described above, or from within the source code of the program slice. The outputs are the formulae displayed in text form as illustrated in Figure 18.

4.7. Graphical user interface

A user interacts with DPUTE through a graphical user interface. Using the interface, a user can select a program to analyse, parse and build the models, and perform all queries supported by the incorporated tools. Typically, a user will first select a variable category to display, and then select one variable from the list displayed in the variable pane. Then, a user will select a category of module relating to the variable chosen. Next, a user will choose one module from the list displayed in the module pane. Finally, a user will select a variety of slices based on the variable and module already chosen, and view the results in the slice pane. For example, a user could first ask for a list of all pure domain input variables, and then select one of them. Then, a user could ask for all modules that assign a value to the pure domain input variable chosen, and select one module from that list.

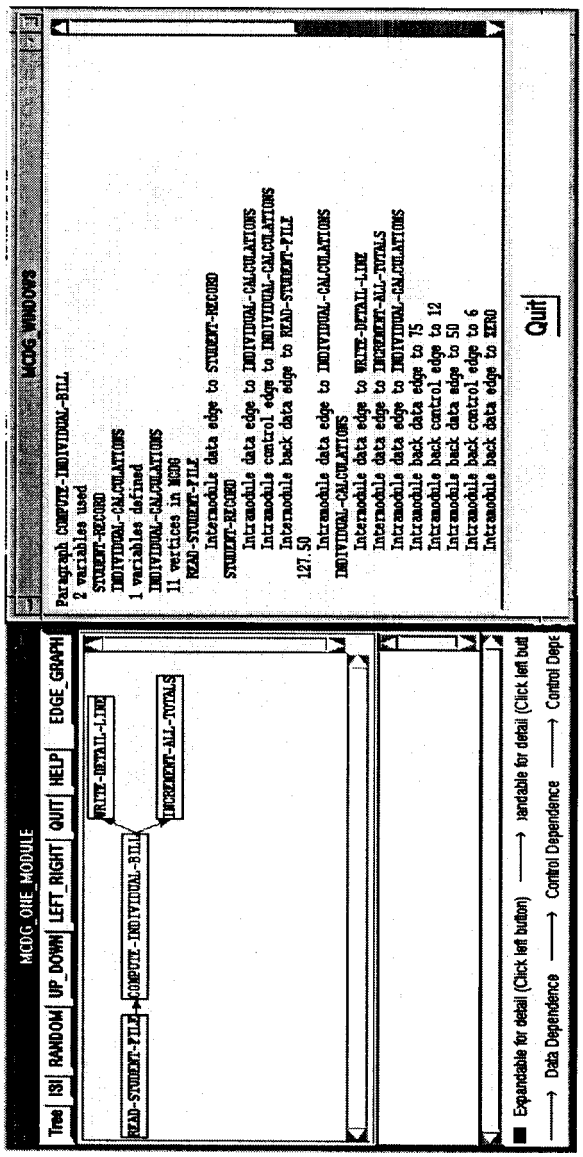


Figure 17. Example of a screen from the MCDG for a paragraph from a program

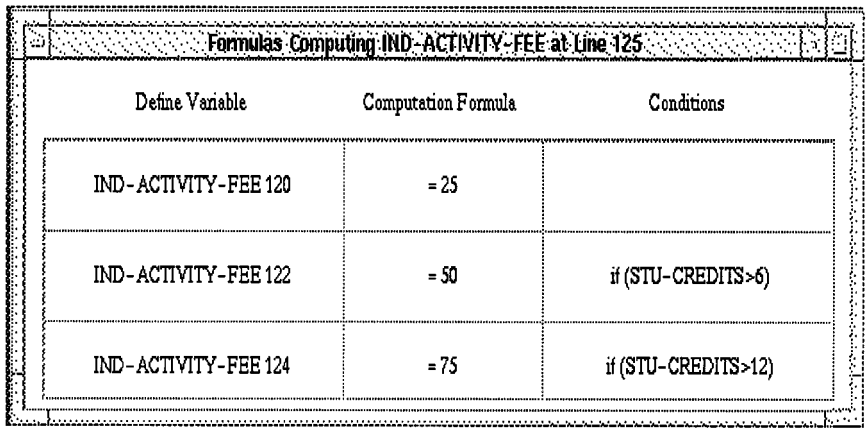


Figure 18. An example of a screen from the formula derivation tool

Finally, a user could select a forward program slice to view the source code that is affected by the selected domain variable starting from that module.

5. EXAMPLE

As Example 8, we now present an interactive business rule extraction scenario for a program that calculates college tuition amounts. Selections from this program have been used earlier in this paper as example material. The program uses two files and eight variables, and consists of eight COBOL paragraphs as detailed in Table 3. The paragraph PREPARE-TUITION-REPORT is the program entry point. The column headed 'Variable' in Table 3 lists all variables referenced in the corresponding paragraphs (procedures) listed in the column headed 'Procedure'. The complete source code can be found in Grauer (1985, pp. 107–109).

Table 3. Summary of tuition example entities

Procedure	Variable
PREPARE-TUITION-REPORT	STUDENT-FILE
READ-STUDENT-FILE	STUDENT-FILE
	STUDENT-RECORD
COMPUTE-INDIVIDUAL-BILL	STUDENT-RECORD
	INDIVIDUAL-CALCUALTIONS
INCREMENT-ALL-TOTALS	STUDENT-RECORD
	INDIVIDUAL-CALCULATIONS
	UNIVERSITY-TOTALS
WRITE-UNIVERSITY-TOTALS	UNIVERSITY-TOTALS
	TOTAL-LINE
	PRINT-LINE

Table 4. Summary of input/output variables for tuition example

Procedure	Input	Output
PROGRAM	STUDENT-FILE HEADING-LINE	PRINT-FILE
PREPARE-TUITION-REPORT	STUDENT-FILE	PRINT-FILE
READ-STUDENT-FILE	STUDENT-FILE	STUDENT-RECORD
COMPUTE-INDIVIDUAL-BILL	STUDENT RECORD	INDIVIDUAL-CALCULATIONS
INCREMENT-ALL-TOTALS	INDIVIDUAL-CALCULATIONS STUDENT-RECORD	UNIVERSITY-TOTALS
WRITE-DETAIL-LINE	STUDENT-RECORD INDIVIDUAL-CALCULATIONS	PRINT-LINE
WRITE-UNIVERSITY-TOTALS	UNIVERSITY-TOTALS	PRINT-LINE

In doing business rule extraction using DPUTE, the first step is to create the syntax tree by parsing the source code. Next, a programmer would use DPUTE to classify variables into useful categories. From the domain variables, DPUTE automatically generates a list of input/output variables for each procedure as well as for the program (see Table 4).

For each input/output variable, DPUTE also generates a list of suggested starting points for forward and backward slicing for the user. Note that the line numbers for input variables are used as starting points for forward slicing while the line numbers for output variables are for backward slicing. These starting points are a list of the statement line numbers in the source code. Table 5 shows the suggested lists for some rows of Table 4.

Suppose we want to know the business rule for the report format of PRINT-FILE. We start backward slicing at line 115 with depth control set, for example, to one. We get the following program slice:

Table 5. Suggested starting points for program slicing

Procedure	Input	Output
PROGRAM	STUDENT-FILE (107) HEADING-LINE (119)	PRINT-FILE (115)
COMPUTE-INDIVIDUAL-BILL	STUDENT-RECORD (134, 135, 140, 142)	INDIVIDUAL-CALCULATIONS (146)
WRITE-DETAIL-LINE	STUDENT-RECORD (155) INDIVIDUAL-CALCULATONS (155)	PRINT-LINE (164)


```
00106  PREPARE-TUITION-REPORT.
00109  PERFORM WRITE-HEADING-LINE.
00111  PERFORM PROCESS-A-RECORD
00112      UNTIL DATA-REMAINS-SWITCH = 'NO'.
00113  PERFORM WRITE-UNIVERSITY-TOTALS.

00118  WRITE-HEADING-LINE.
00119  MOVE HEADING-LINE TO PRINT-LINE.
00120  WRITE PRINT-LINE
00121      AFTER ADVANCING PAGE.

00127  PROCESS-A-RECORD.
00130  PERFORM WRITE-DETAIL-LINE.

00154  WRITE-DETAIL-LINE.
00163  MOVE DETAIL-LINE TO PRINT-LINE.
00164  WRITE PRINT-LINE
00165      AFTER ADVANCING 1 LINE.
00167  WRITE-UNIVERSITY-TOTALS.
00173  MOVE TOTAL-LINE TO PRINT-LINE.
00174  WRITE PRINT-LINE
00175      AFTER ADVANCING 2 LINES.
```

From the above program slice, we easily get the following business rule for the report format as follows:

1. print a single heading line at the start of the report,
2. print a detail line for each record read, and
3. print a total line at the end of the report.

Then, we want to know the format of the detail line. We start backward slicing at line 164 and variable PRINT-LINE. We also limit the slicing space within the procedure WRITE-DETAIL-LINE. We get the program slice and present it in its formula view:

```
PRINT-LINE = DETAIL-LINE ( STU-NAME,
                           STU-SOC-SEC-NO,
                           IND-TUITION,
                           IND-UNION-FEE,
                           IND-ACTIVITY-FEE,
                           STU-SCHOLARSHIP,
                           IND-BILL
).
```

In the above formula, we see that DETAIL-LINE has seven fields printed in the order as listed above. Three of the seven fields, STU-SOC-SEC-NO, STU-SCHOLARSHIP,

Table 6. The business rule for the ACTIVITY-FEE calculation

Activity fee	Credits
$25	6 or less
$50	7 to 12
$75	more than 12

STU-NAME, are fields in a STUDENT-RECORD that is input from the STUDENT-FILE. Thus, the next step is to explore how the values of the other four fields, IND-TUITION, IND-UNION-FEE, IND-ACTIVITY-FEE and IND-BILL, are calculated. We know they are fields in INDIVIDUAL-CALCULATIONS. Again, suppose we want to know how to calculate the value of IND-ACTIVITY-FEE. We start backward slicing at line 146 and variable IND-ACTIVITY-FEE, then we get the following program slice:

```
00139  MOVE 25 TO IND-ACTIVITY-FEE.
00140  IF STU-CREDITS > 6
00141       MOVE 50 TO IND-ACTIVITY-FEE.
00142  IF STU-CREDITS > 12
00142       MOVE 75 TO IND-ACTIVITY-FEE.
```

The program slice above can be easily translated to the business rule, with the results as shown in Table 6.

In a similar way, we can get business rules for the calculation of the three variables IND-UNION-FEE, IND-TUITION, and IND-BILL.

6. CONCLUSION

In this paper, we have presented a procedure and a tool set for business rule extraction (BRE). Many prasticing software maintainers we interacted with have expressed the need for such procedures and tool sets, and the importance of and criteria for BRE. The most important criterion is that the business rules extracted must reflect the true state of the software, no matter whether the software is structured or otherwise. We have presented several BRE techniques including variable classification, program slicing, slicing criteria selection, slicing space reduction, rule representation and dependence browsing. These techniques form the core of the BRE tools that can be used by maintainers to extract business rules interactively.

References

Abbattista, F., Lanubile, F. and Visaggio, G. (1993) 'Recovering conceptual data models is human intensive', in *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering (SEKE 93)*, San Francisco CA, Knowledge System Institute, Skokie IL, pp. 534–543.

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools*, Addison–Wesley Publishing Co., Reading MA, 796 pp.
Chapin, N. (1978) 'Input/output tables in structured design', in *Structured Analysis and Design, Volume 2*, Infotech International Ltd., Maidenhead, Berkshire, UK, pp. 43–55.
Chen, S., Heisler, K. G., Tsai, W.-T., Chen, X. P. and Leung, E. (1990) 'A model for assembly program maintenance', *Journal of Software Maintenance*, **2**(1), 3–32.
Chen, S., Tsai, W.-T. and Chen, X. P. (1992) 'SAMEA: an object–oriented environment for supporting assembly programs', *Journal of Software Engineering and Knowledge Engineering*, **2**(2), 197–226.
Chen, X. P., Tsai, W.-T., Joiner, J. K., Gandamaneni, H. and Sun, J. (1994) 'Automatic variable classification for COBOL programs', in *Proceedings of COMPSAC 94*, Taipei, Taiwan, IEEE Computer Society Press, Los Alamitos CA, pp. 432–437.
Chen, X. P., Tsai, W.-T., Huang, H., Poonawala, M., Rayadurgam, S. and Wang, Y. (1996) 'Omega—an integrated environment for C++ program maintenance', in *Proceedings of the International Conference on Software Maintenance*, Monterey CA, IEEE Computer Society Press, Los Alamitos CA, pp. 114–123.
Ferrante, J., Ottenstein, K. J. and Warren, J. D. (1987) 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, **9**(3), 319–349.
Grauer, R. T. (1985) *Structured COBOL Programming*, Prentice–Hall, Englewood Cliffs NJ, 426 pp.
Gopal, R. (1991) 'Dynamic program slicing based on dependence relations', in *Proceedings of the Conference on Software Maintenance—1991*, Sorrento, Italy, IEEE Computer Society Press, Los Alamitos CA, pp. 191–200.
Hecht, M. S. (1977) *Flow Analysis of Computer Programs*, Elsevier North–Holland, New York NY, 232 pp.
Heisler, K. G., Kasho, Y. and Tsai, W.-T. (1993) 'A reverse engineering model for C programs', *Information Sciences*, **68**(1–2), 1–55.
Heisler, K. G., Tsai, W.-T. and Powell, P. (1989) 'An object–oriented software–oriented maintenance model', in *Proceedings of COMPCON Spring 89*, San Francisco CA, IEEE Computer Society Press, Los Alamitos CA, pp. 248–253.
Horwitz, S., Reps, T. and Binkley, D. (1990) 'Interprocedural slicing using dependence graph', *ACM Transactions on Programming Languages and Systems*, **12**(1), 26–60.
Huang, H., Tsai, W.-T. and Subramanian, S. (1996) 'Generalized program slicing for software maintenance', in *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE 96)*, Lake Tahoe NV, Knowledge Systems Institute, Skokie IL, pp. 261–268.
Joiner, J. K. and Tsai, W.-T. (1993) 'Ripple effect analysis, program slicing and dependence analysis', Technical Report, TR 93–84, Department of Computer Science, University of Minnesota, Minneapolis MN 55455, 23 pp.
Joiner, J. K., Tsai, W.-T., Chen, X. P., Subramanian, S., Sun, J. and Gandamaneni, H. (1994) 'Data–centered program understanding', in *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia Canada, IEEE Computer Society Press, Los Alamitos CA, pp. 272–281.
Joiner, J. K. and Tsai, W.-T. (1997) 'Reengineering legacy COBOL', *Communications of ACM*, to appear.
Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, M. (1981) 'Dependence graphs and compiler optimizations', in *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, Williamsburg VA, Association for Computing Machinery, Inc., New York NY, pp. 207–218.
Ong, C. L. and Tsai, W.-T. (1993) 'Class and object extraction from imperative code', *Journal of Object–Oriented Programming*, **6**(2), 58–60, 68.
Sano, T., Tsai, W.-T. and Rayadurgam, S. (1997) 'Interview with Takashi Sano', *Journal of Software Maintenance*, **9**(4), 253–268.
Tsai, W.-T. (1997) 'Application of data–centered approach to year 2000 problem', in *Proceedings of COMPSAC'97*, Washington DC, IEEE Computer Society Press, Los Alamitos CA, pp. 287–288.

Tsai, W.-T., Kirani, S., Lee, H. J., Heisler, K. G., Subramanian, S., Joiner, J. K., Mojdehbakhsh, R., Boddu, C., Wen, I. and Bhattacharya, S. (1993) 'An integrated model for reverse engineering and its implications to forward engineering', Technical Report, TR 93–40, Department of Computer Science, University of Minnesota, Minneapolis MN 55455, 32 pp.

Wang, Y., Tsai, W.-T., Chen, X. P. and Rayadurgam, S. (1996) 'The role of program slicing in ripple effect analysis', in *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE 96)*, Lake Tahoe NV, Knowledge Systems Institute, Skokie IL, pp. 269–376.

Weiser, M. (1982) 'Programmers use slices when debugging', *Communications of ACM*, **25**(7), 446–452.

Weiser, M. (1984) 'Program slicing', *Transactions on Software Engineering*, **SE–10**(4), 352–357.

Yau, S. S., Collofello, J. S. and MacGregor, T. (1978) 'Ripple effect analysis of software maintenance', in *Proceedings of COMPSAC'78*, Chicago IL, IEEE Computer Society Press, Los Alamitos CA, pp. 60–65.

Authors' biographies:

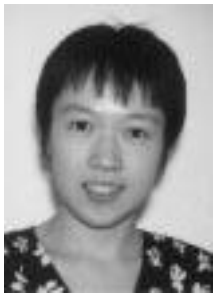
Hai Huang is a Ph.D. candidate in the Department of Computer Science, University of Minnesota. He received a B.Sc. and an M.Sc. in Computer Science and Technology from the University of Science and Technology of China, and an M.Sc. in Computer Science from the University of Minnesota. His main research interests include software maintenance, software reverse engineering and software engineering. E-mail: hhuang@cs.umn.edu

Wei-Tek Tsai is a Professor of Computer Science and Engineering at the Department of Computer Science, University of Minnesota. He received an S.B. in Computer Science and Engineering from the Massachusetts Institute of Technology, an M.Sc. and a Ph.D. in Computer Science from the University of California, Berkeley. His main research interests include software engineering, internet and computer systems. E-mail: tsai@cs.umn.edu

Sourav Bhattacharya is an Associate Professor of Computer Science at Arizona State University, Tempe, Arizona. He received his Ph.D. in Computer Science from the University of Minnesota in 1993. His research interests include high-assurance systems, system engineering, and parallel and distributed systems. E-mail: sourav@asu.edu

Xiaoping P. Chen is currently a member of the technical staff in Lucent Technologies, Inc. He received his Ph.D. in Computer Science from the University of Minnesota in 1996. His interests include object-oriented analysis, design and development methods, telecommunications, and graphical user interfaces. He has published several papers on software maintenance and reverse engineering.

Yamin Wang is currently a Ph.D. candidate in the Computer Science Department, University of Minnesota. He received his B.S. and M.S. from the Department of Computer Science and Technology of Tsinghua University, Beijing, China. His main research interests include object-oriented design and testing, software maintenance, and software engineering.



Jianhua Sun is a Ph.D. student in the Department of Computer Science, University of Minnesota. She received a B.Sc. in Computer Science and Technology from the University of Science and Technology of China, and an M.Sc. in Computer Science from the University of Minnesota. E-mail: sun@cs.umn.edu

